5-2014

The Rusterizer: An Art-Directable and Semi-Procedural Tool for Generating Rust Surfaces

Karen Stritzinger
Clemson University, kmstrit@gmail.com

The Rusterizer: An Art-Directable and Semi-Procedural Tool for Generating Rust Surfaces

---

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Fine Arts
Digital Production Arts

---

by
Karen Stritzinger
May 2014

Accepted by:
Dr. Jerry Tessendorf, Committee Chair
Dr. Andrea Feeser
Dr. Sophie Joerg

Abstract

The Rusterizer is an art-directable tool designed to facilitate the creation of rust on the surfaces of 3d models. Cellular automata are combined with a procedural shader to create an animated growth effect. A workflow for using the tool is provided. The result is demonstrated with a time-lapse animation of a robot in an alley as it rusts. Future extensions and improvements to the tool are suggested.

Dedication

I dedicate this work in memory of the life of my dear friend, Cory James Payne.

"Let me be your everlasting light,

the sun when there is none."

-The Black Keys

Acknowledgments

First I would like to thank my advisor, Dr. Jerry Tessendorf, for pushing me to extend the scope of my thesis. Had he not suggested I create a method of generating rust procedurally, I would probably not have gone to the trouble to learn RenderMan or to learn how to build a tool for others to use. I also want to thank Dr. Tessendorf for strengthening the Digital Production Arts Program and helping create a pipeline that has taken our work to a new level. I want thank Sabrina Riegel for her generous feedback during the development of the Rusterizer, as well as during the production of "QA-ARM-A." I have been given a new perspective on the surfacing process that would not have been possible otherwise. I thank Tiffany Feeney for helping to coordinate my thesis defense with Sabrina. I thank Dr. Andrea Feeser and Dr. Sophie Joerg for being a part of my thesis committee and for their time spent reviewing my work and giving me their thoughts. I thank Dr. Juan Gilbert for providing my assistantship during the entirety of my graduate education and part of my undergraduate studies. I truly appreciate the opportunity I was given to be a part of the creative team in the Human-Centered Computing Lab. I thank Kacey Coley for his frequent words of encouragement when I doubted myself. I also thank Kacey for his help with setting up the Rusterizer GUI, distributing the tool, and general troubleshooting. I thank Jon Barry for surfacing and modeling several props that were used in "Life After QA-ARM-A." I thank Zhaoxin Ye for reading my thesis and giving me feedback. I also want to thank any student who attended weekly meetings for their critique. I thank my parents not only for allowing, but encouraging me to pursue my passions. My mother has been a constant inspiration for the type of person I would like to become, and her natural creativity has always amazed me. My father has always been there to give me reassurance and advice when I was worried, and he made me feel that I was capable of anything I wanted to do, no matter how challenging. I also thank my sisters, Kelly and Christina Stritzinger, for supporting me. I wouldn't be here without my family, and I love them all very much.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Figure 1.1: Still frame from "QA-ARM-A."

The Rusterizer is a surfacing tool inspired by challenges encountered during the production of the animated short titled "QA-ARM-A." The short depicts a quality assurance robot who is having a bad day at work. The term "surfacing" refers to the process of creating color and textures for 3d models. This process includes the creation of hand-painted textures and the development of shaders, which are sometimes referred to as materials. Shaders are assigned to 3d models to determine the way that light interacts with the surface, and they also control the way that texture maps and procedural patterns are applied. Many standard shaders are available in 3d software such as Maya, although it is often desirable to write shaders from scratch to achieve superior control.

"QA-ARM-A" was created during the DreamWorks/Digital Production Arts Summer Program in 2013 at Clemson University. A team of five students worked together for ten weeks under the guidance of DreamWorks mentors. Concept artwork and some basic requirements were given. The story was to be based on a simple prompt: "A robot seeks to replace a missing part when something unexpected occurs." Some restrictions were imposed in order to allow a high level of refinement in all areas of production. The length of the short was limited to 15 seconds, the story needed to be a vignette rather than a full narrative, and only hard-surface characters were allowed. A layered effect was required along with photo-realistic lighting and surfacing. Finally, the quality of the result was expected to be professional and artifact free, rather than a student level production.

The first stage of production called visual development involved research, character design, as well as creation of inspirational artwork. This stage continued through the storyboarding process in order to make any necessary design changes to support the plot. The overall design of "QA-ARM-A" was meant to feel older, and the robot character was imagined to be somewhat of a Luddite. Although an electronic system controls the movement of the track, the robot still uses a pencil and clipboard to make notes. The character's surface color and textures were inspired by tin toy robots that were created in the 50's, as seen in Figure 1.2.

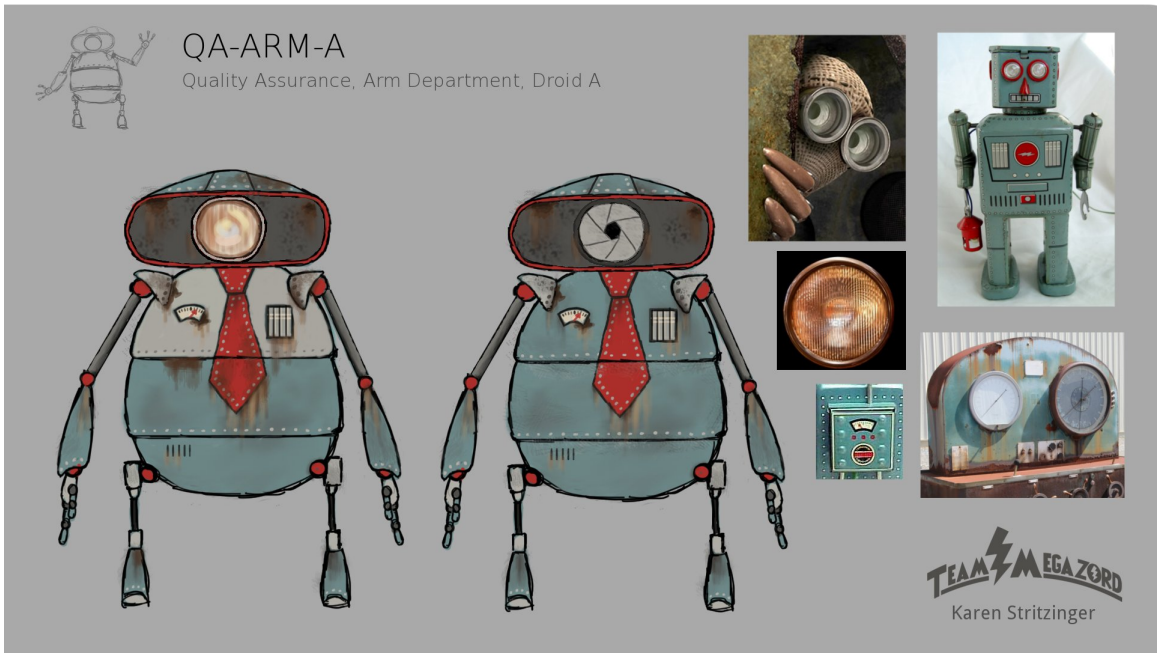


Figure 1.2: "QA-ARM-A" character surfacing design.

When visual development, storyboarding, as well as character and environment modeling were complete, it was time for surfacing to begin. A render of the surfaced robot is shown in Figure 1.3. In order to create a realistic sense of dated materials, several layers of wear and tear were required on the surfaces. The design of the robot's surface called for a significant amount of rust. The texture layers for the rust included manipulated photographs of rust, a chipped paint edge, a bumpy texture and specular irregularities, as shown in Figure 1.4. A semi-transparent drip pattern was painted below large spots of rust where water would flow down the surface.

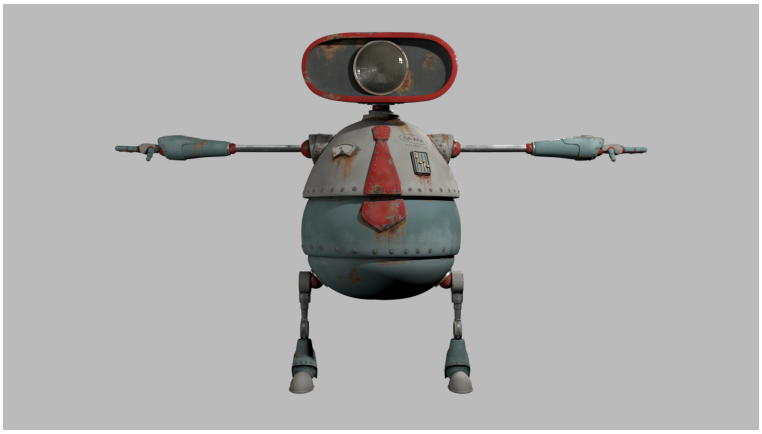
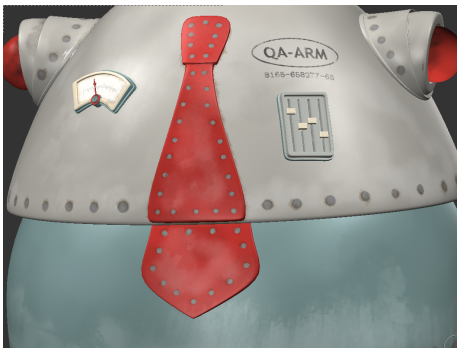
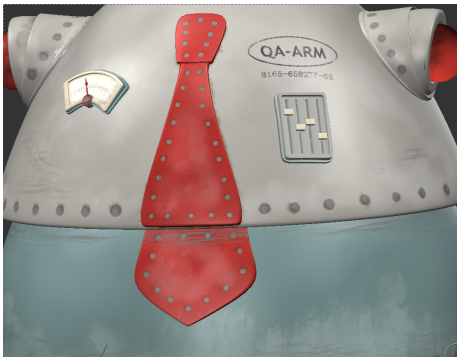


Figure 1.3: Render of QA-ARM-A with original hand-painted rust textures.

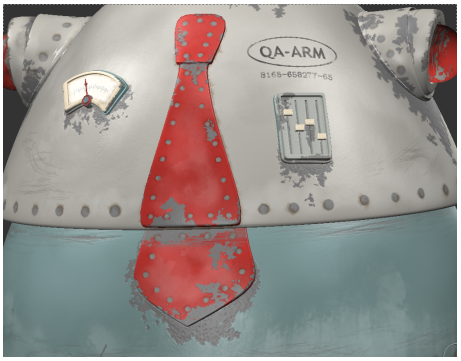
In order to paint the rust layers manually, the 3d texture painting software Mari was chosen over Photoshop. Photoshop is the software traditionally used for painting textures, but there are some limitations to this tool. In Photoshop it is difficult to visualize what the result of the textures will look like on a 3d model in Maya, and a good deal of time is spent switching between the 2d and 3d applications to ensure that everything is working properly. Mari was used to paint the robot's textures because it greatly reduced the time needed for the process by allowing the user to paint directly on the surface of a model. The software also allows the user to paint in 2d UV layout space if necessary. Texture seams are not a significant issue and previews of specular, bump and displacement maps are available. Despite the improved painting workflow with Mari, there were still some aspects of the surfacing process that could be improved. During iterations of surfacing, it was often necessary to vary the size of the rust spots or to move them among locations on the surface of the character to satisfy the art direction. The process of editing all of the rust texture layers for several iterations became tedious, and this provided the motivation to create the Rusterizer.



(a) Base



(b) Scratches



(c) Chipped Paint and Bump

(d) Rust

(e) Specular Irregularity

(f) Rust Drip

Figure 1.4: Rust texture layers.

The Rusterizer was designed to facilitate the process of creating rust spots on the surface of a 3d character, prop, or other model. The tool provides control over the age of rust, generating an animated growth effect. The Rusterizer is meant to be used with the Digital Production Arts (DPA) pipeline structure, but could be adapted for other pipelines in the future. The Rusterizer was also designed to be artist-friendly; a tool that someone with a less technical background would feel comfortable using. A graphic user interface (GUI) was created for use in Maya to help make the program more approachable. Finally, the tool was created to be art-directable so that quick changes in size or location of rust spots would be feasible.

Several features were included in order to create a high level of control for the artist. These features include two painted input maps that are used to generate the rust growth patterns. One of these maps, the "Rust Distribution Map,"is a grayscale map that designates the areas on the model where the user would like rust spots to initially appear on the surface. This map is used as the starting state of a cellular automata simulation, and is required for the tool to work. An optional "Rust Probability Map" is another grayscale map that designates the likelihood that rust will grow on a particular part of the surface. If this map is omitted, the Rusterizer will use default probability values to simulate the rust growth patterns.

The input maps drive a cellular automata simulation which plugs into the Rusterizer Shader in Maya, along with four procedural noise and rust color presets. These presets provide a starting point that the user is able to modify manually in Maya's attribute editor, should they so desire. The growth of the rust can also be controlled by animating an attribute called "Age of Rust" in the Rusterizer Shader. If the production calls for static rust, the growth attribute can be used to quickly change the size of rust spots if such a change were required at any point in the look development process. If an artist would like to add hand-painted details to the rust, the Rusterizer Shader's noise patterns can be transferred from Maya to a texture painting application by using the Map Export mode of the Rusterizer Shader.

Painting rust textures strictly by hand allows for very direct control of the shape and size of rust patterns independent of one another. This method is ideal for a character or prop that will be close to the camera for the majority of a production, when the ability to make very specific changes is necessary. Despite the advantage of control, completely hand-painted textures are time consuming. Manually animating the growth of rust on a surface over time would be sufficiently time-intensive to become burdensome for a production.

One of the main advantages of using the Rusterizer over a manual texture painting method would be that the color and the noise patterns of the rust can be changed instantaneously. Animating the growth of rust over time is automatic, although it costs some simulation time. The disadvantage of using the Rusterizer to create animated surfaces is that the size of rust spots can be controlled as a whole, but individual control for the size of specific spots is not possible unless the tool is used for static rust. For example, a texture artist might use the Rusterizer to determine the general size of the rust spots that they desire. Once this has been chosen, they would export the result into texture maps in order to complete the rest of the details manually. This method allows the Rusterizer to do most of the work, creating a base for painting so that less time will need to be spent adding details by hand.

Chapter 2

Background

In "Modeling and Rendering of Metallic Patinas," Dorsey and Hanrahan discuss a phenomenological model for the creation of metallic patinas on surfaces [4]. Patinas are the addition or removal of material on a surface due to a chemical reaction. They use the Kubelka-Munk model to simulate reflection and transmission through a layered surface. Their model can also represent the appearance of metals over time, and the result is demonstrated with a set of copper objects, as shown in Figure 2.1. The process differs from the Rusterizer in that it does not allow for an artist to control the location of the surface details. The result looks realistic enough to be useful for a production that requires a photoreal style.

Figure 2.1: Aging copper statuette.

Lipton, Museth and Sutherland discuss their frost effect in "Jack's Frost: Controllable Magic Frost Simulations for 'Rise of the Guardians'" [10]. The frost was achieved with the use of a colored input map to control the shape and speed of the frost's growth as shown in Figure 2.2. The input map color is used to determine a search radius for an unstructured cellular automata simulation.

In the unstructured cellular automata simulation, points will turn on when a neighbor point within the search radius is also on. This means that a larger search radius will allow the growth speed to increase because it is more likely to encompass other neighboring points that are on. A velocity field is created from the cellular automata simulation which is used in Houdini to advect thousands of points, leaving trails on a surface. The result from Houdini is rendered as curves and used in Nuke for the final adjustments. The use of cellular automata in this work helped to inspire a similar use of the simulation method in the Rusterizer.

Figure 2.2: Input map and render of Jack's Frost, copyright DreamWorks Animation.

In addition to the previously mentioned work, some existing rust surface techniques proved to be influential during the development of the Rusterizer. Paul Kanyuk, a Technical Director at Pixar, created an aging demonstration with a realistic procedural rust shader in Slim, RenderMan's node-based material editor [7]. His work provided an example of what is possible with a procedural method. The paint chipping, shown in Figure 2.3, looks very realistic and served as a goal to strive towards during the development of the Rusterizer Shader.

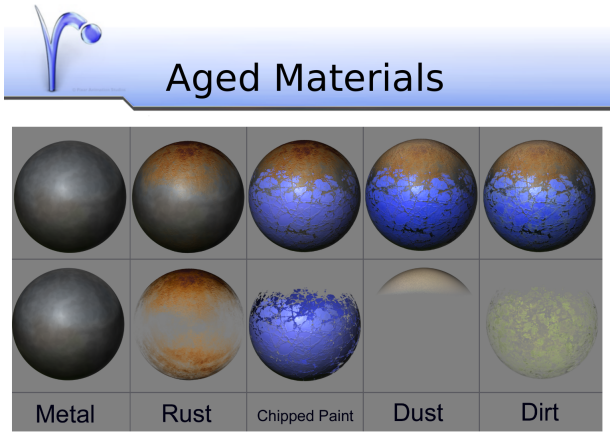


Figure 2.3: Rust Shader by Paul Kanyuk.

Student Katie Green created a procedural shader, seen in Figure 2.4, that uses an input map to determine the shape of rust spots [5], although the method does not provide a way to animate the growth. The result also uses displacement, but does not look photoreal. The work was useful as an example of a shader that used an input map.

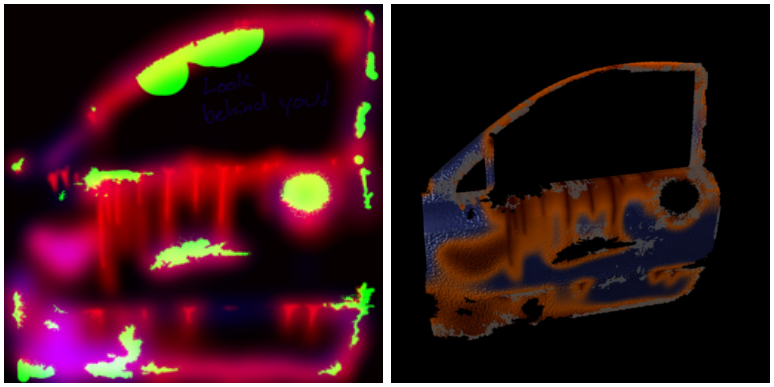


Figure 2.4: Rust Shader by Katie Green.

Another student, Lauren Perry, created a shader that allows the user to animate the rust coverage to increase over time [11]. The tool she created also combines the procedural rust with rust drip patterns. These are areas where the color of the rust is spread by the flow of water across the surface. The rust color can also spread to other objects, such as the ground plane as shown in Figure 2.5. The combination of the rust drip pattern with the procedural, animated rust patterns provided another goal for the Rusterizer to aspire to.

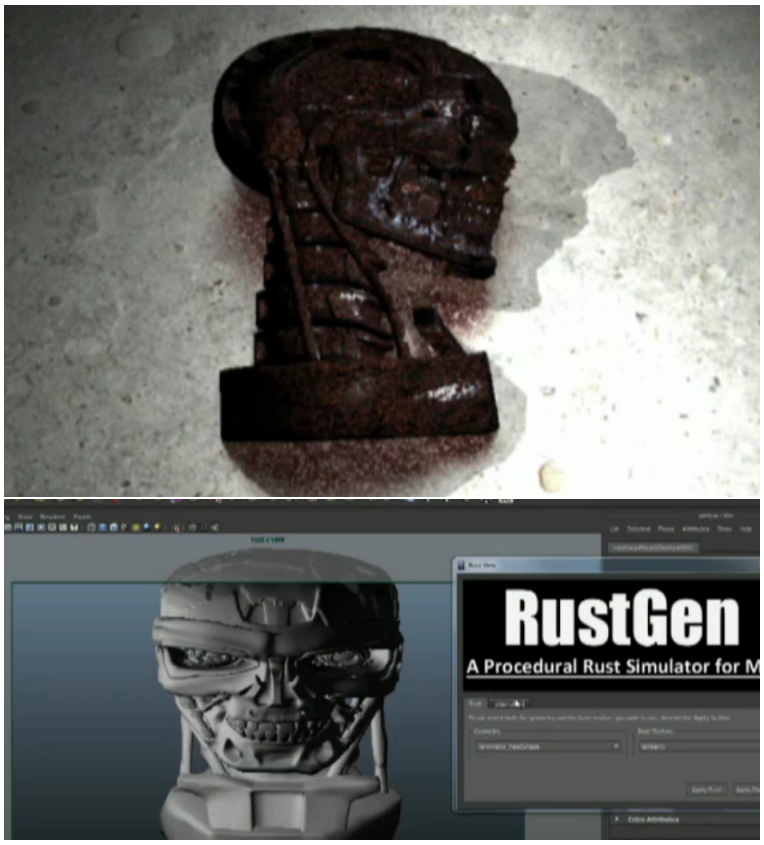


Figure 2.5: Rust Shader by Lauren Perry.

Early in the research process, cellular automata were chosen for use in creating rust growth. Cellular automata are models that be can be used to simulate natural growth patterns. They consist of a structure, often a grid, of cells and a set of rules that these cells must follow. Automata can be one, two and even three dimensional. Most cellular automata simulations involve a series of iterations in which cells are either on or off, although more than two states are possible. In the case of two possible states, cells use the set of rules relating to the previous state of their neighbors to determine whether or not they should turn on or off in the next iteration.

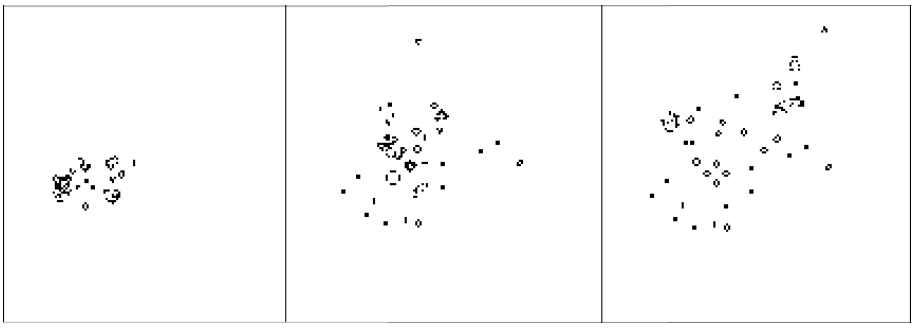


Figure 2.6: Conway's Game of Life.

One popular type of cellular automata, created by John Horton Conway [12], is called the Game of Life. A series of iterations of the Game of Life are shown in Figure 2.6. The Game of Life uses the Moore neighborhood definition [12]. In the Moore neighborhood the current cell's neighbors are defined to include any of the eight adjacent cells to the current cell, as seen in Figure 2.8. In this diagram, the green cell represents the current cell and the neighbor cells are blue. The von Neumann neighborhood, also seen in Figure 2.8, is another neighborhood definition that is used in other automata. The cell rules for Conway's Game of Life are shown in Figure 2.7.

```
if (previous cell state = off AND 3 or more neighbors are on)
    current cell state = on
else if (previous cell state = on AND 2 or 3 neighbors are on)
    current cell state = on
else
    current cell state = off
```

Figure 2.7: Conway's Game of Life Cellular Automata Rules.

(a) The Moore neighborhood. (b) The von Neumann neighborhood.

Figure 2.8: Types of cell neighborhoods.

Figure 2.9: Rusterizer Cellular Automata Simulation.

Figure 2.9 shows a series of iterations of the Rusterizer cellular automata simulation. The simulation also uses the Moore neighborhood. For comparison with the Game of Life, the rules for the cellular automata in the Rusterizer are shown in Figure 2.10.

```
age increment = (1.0/number of frames)*5.0
probability A = probability map value
probability B = (probability A - ((1.0-probability A)/2.0))
if (current frame is first frame)
    current cell state = distribution map value
    current age = 0
if (previous cell state = on)
    current cell state = previous cell state
    current age = previous age + age increment
else if (previous cell state = off AND 3 neighbors are on AND a random number >= probability A)
    current cell state = on
else if (previous cell state = off AND 4 neighbors are on AND a random number >= probability B)
    current cell state = on
if(current age > 1)
    current age = 1
```

Figure 2.10: Rusterizer Cellular Automata Rules.

Chapter 3

Workflow

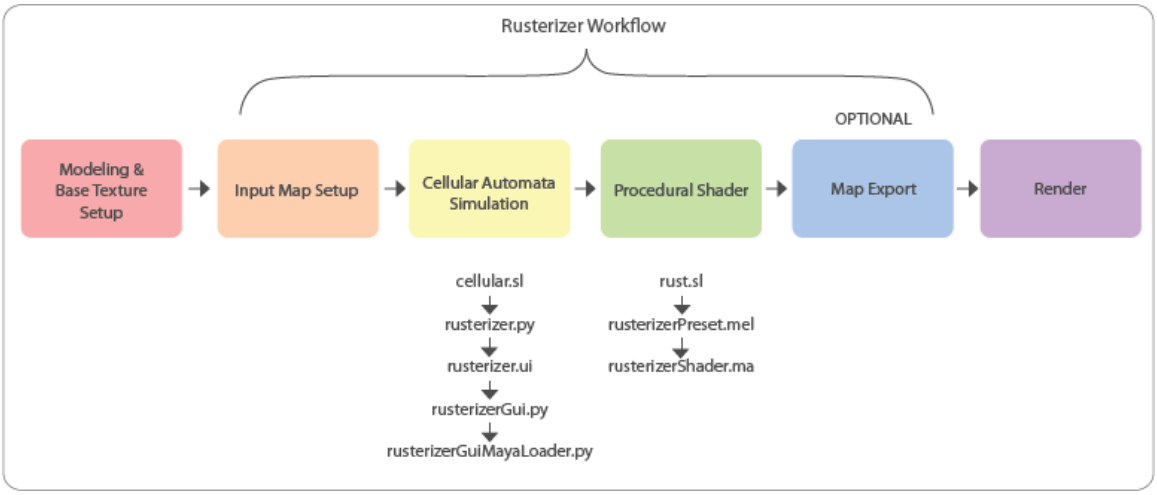


Figure 3.1: Diagram showing Rusterizer workflow in relation to the rust surfacing process.

The Rusterizer workflow involves three main steps: input map setup, cellular automata simulation, and application of the simulation output in a procedural shader. After the third step, there is an optional fourth step involving the manual exporting of texture maps from the procedural shader into an image editing software. Figure 3.1 is a diagram depicting the relationship of the various elements that come together to form the Rusterizer tool.

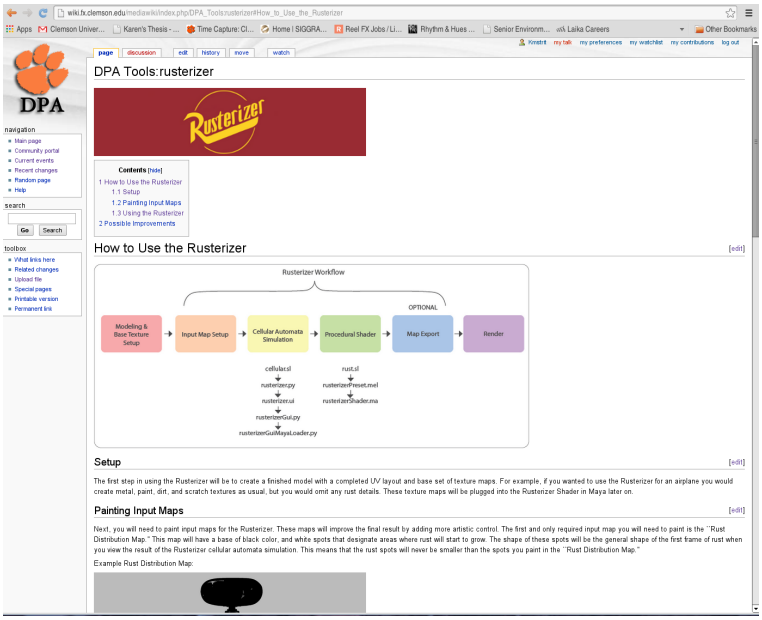


Figure 3.2: Screenshot of the Rusterizer wiki page.

To help artists encountering the Rusterizer for the first time, a help button labeled "?" in the GUI links directly to a DPA wiki page, and covers the steps required to use the tool from start to finish. The wiki page also describes possible ways to improve or modify the tool through the creation of new shaders. Figure 3.2 shows this wiki page. The following workflow assumes that the tool is being used in the DPA pipeline. Many of the steps may be different if it were to be used elsewhere.

The first step required to use the Rusterizer is to create a finished model with a completed UV layout and base set of texture maps. For example, if an artist wanted to use the Rusterizer for an airplane they would create metal, paint, dirt and scratch textures as usual, but would omit any rust details. These texture maps will be plugged into the Rusterizer Shader in Maya later on.

A linear color space is recommended for the Rusterizer. In the Rusterizer Shader, base textures are assumed to be sRGB. The shader uses default gamma correction of .4545 to convert sRGB values to linear. If a texture image already happened to be linear, this value could be set to 1 instead. The "Color Gamma" attribute in the Rusterizer Shader allows direct control for this. Render settings should be set such that the image renders linear, although it displays in sRGB.

The next step in the process is to paint input maps for the Rusterizer. These maps will improve the final result by adding more artistic control. The only required input map is the "Rust

Distribution Map." This map has a base color of black, and white spots that designate areas where rust initially grows. The shape of these spots will be the general shape of the first frame of rust after the simulation is complete and images are rendered with the procedural shader. This means that the rust spots will never be smaller than the spots painted in the "Rust Distribution Map." Figure 3.3 shows an example of this map.

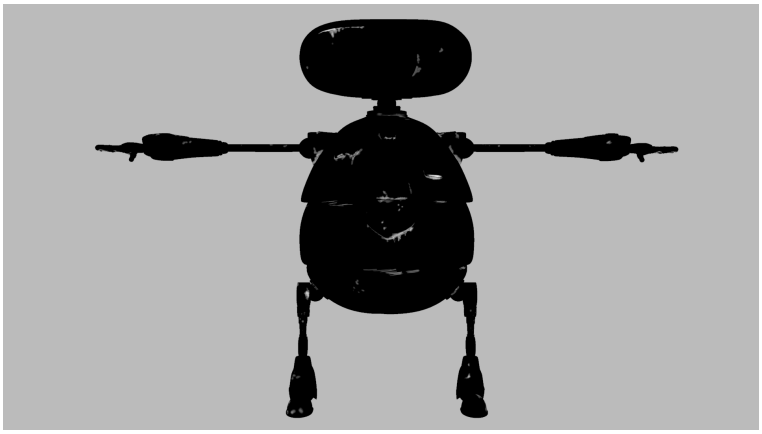


Figure 3.3: Example "Rust Distribution Map"

When painting a "Rust Distribution Map" for a model with multiple UV patches, it is important to keep the flow of rust between patches in mind. Each patch requires its own simulation, and this can make the transition between patches more noticeable in the final result. A way to combat this problem is to consciously paint white spots in the map that span the transition between patches. This will allow rust to grow on both patches in a similar way, rather than hoping that they don't cross a patch border later on. An example of this might be a character that has a UV patch for each limb, and other patches for the torso and head. The edges between the torso and the limbs are prone to abrupt changes in surface color from rust to paint. An artist might paint a few white spots over the edge between the limbs and the torso to reduce the chance for these issues to occur. In Figure 3.4, several examples of this technique are shown. The red lines represent borders between patches, and several white spots are painted across these lines.

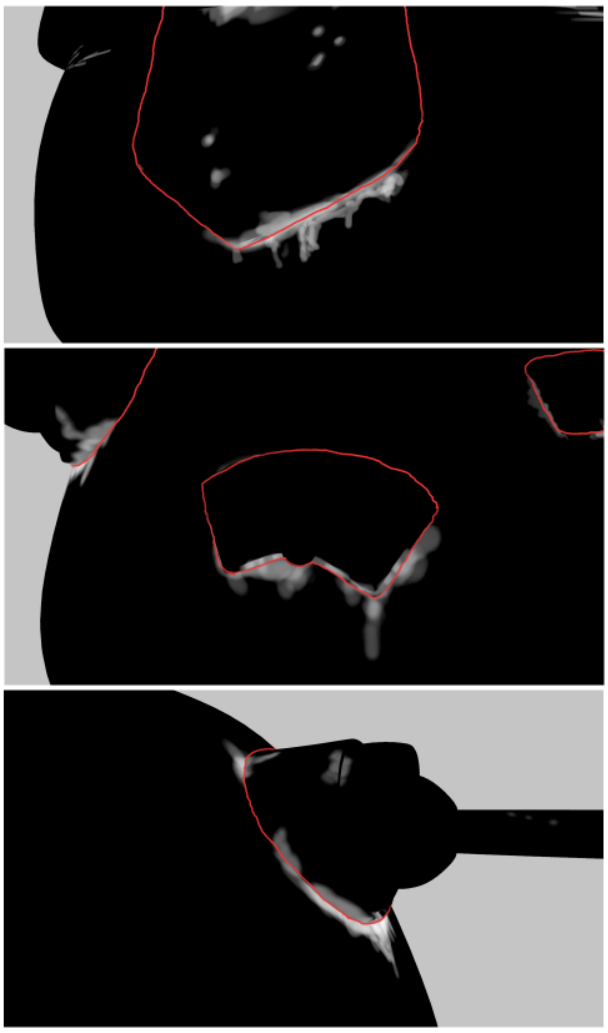


Figure 3.4: Example of patch edge paint transitions.

Another map that can alter the result of the Rusterizer is called the "Rust Probability Map." This map uses values from 0-1, or black to white, to determine the probability that rust will grow at any particular point on the surface. This means that areas painted true black will prevent any rust from growing, while areas with white will encourage rust to grow faster. Figure 3.5 shows an example of this map. Rust is the process of iron oxidation that occurs when iron combines with water and oxygen [1]. Therefore, rust is more likely to grow in areas where moisture accumulates. This means that if there are any crevices on an object, these areas will usually have more rust than other areas.

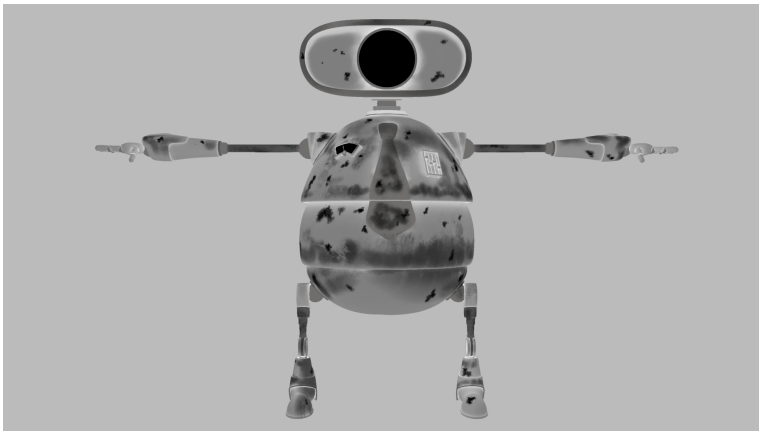


Figure 3.5: Example "Rust Probability Map"

One way to approximate this effect is by adding a render of ambient occlusion (AO) to the "Rust Probability Map." AO refers to a method in computer graphics of approximating how exposed points on a model are to diffuse light [9]. An example of this is the subtle contact shadows that appear between an oject and the surface beneath it. The AO information can be inverted so that the black areas are white, and this can be used as a layer in the map to increase the odds of rust growth. Mari makes this process easier because it provides a function to calculate AO. Once the AO has been calculated in Mari, there is also an "Ambient Occlusion Mask" that can be inverted and used to paint a map representing the information.

Other details like scratches and dents may be good things to include. This map is a trial and error process, and helps to give the rust a more varied, organic look. If the "Rust Probability Map" is omitted, the Rusterizer will calculate the rust growth using default values of probability. Figure 3.6 shows the rust result based on the input maps from Figures 3.3, 3.4 and 3.5.

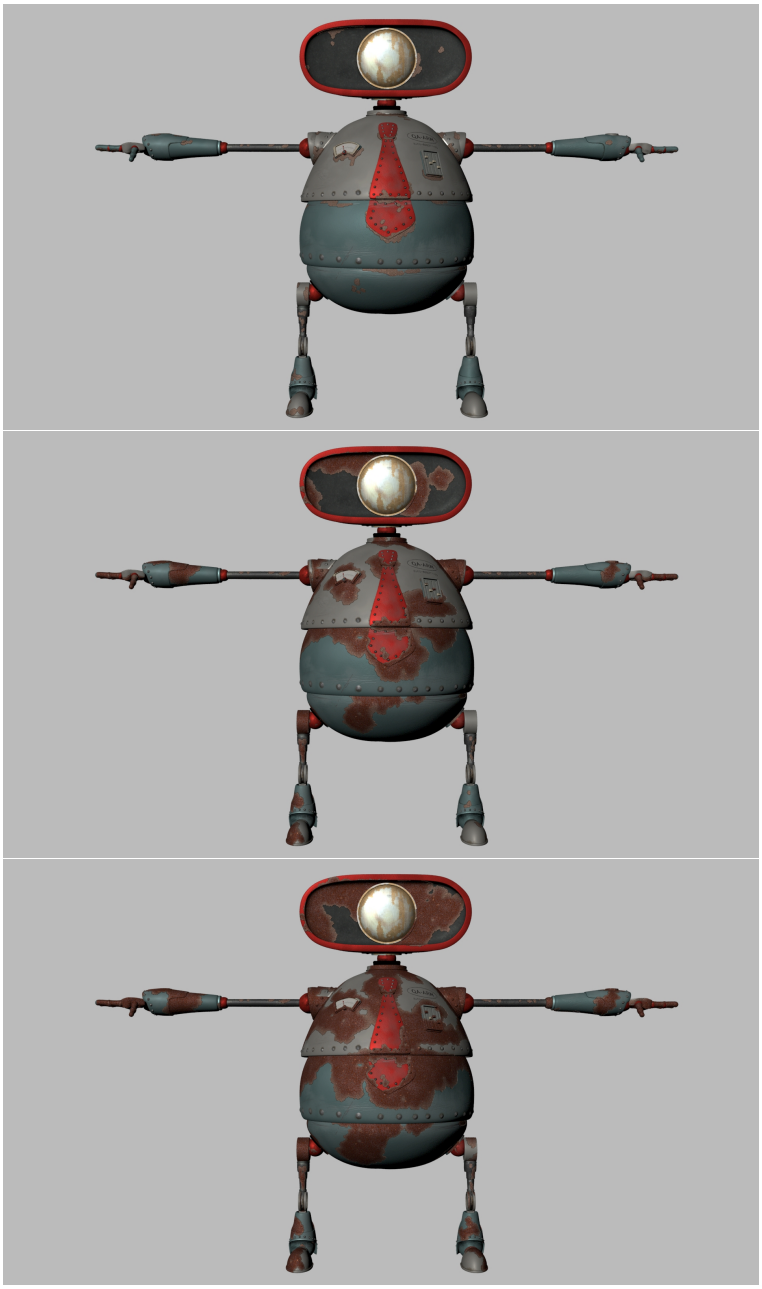


Figure 3.6: Rendered rust result from example Rust Distribution and Probability maps.

Figure 3.7: Rusterizer Shelf Icon

Once the input maps have been completed, the Rusterizer launches in Maya from the DPA shelf "Rust" icon shown in Figure 3.7. The Rusterizer GUI has several browse buttons to locate the input maps that were previously created, as well as to set the simulation output location, as shown in Figure 3.8. The path for the cellular automata shader has a default setting according to the current DPA pipeline. This path can be changed to a new location if the user would like to make their own adjustments to the cellular automata shader, and this is discussed in more detail in Chapter 6. If the model has multiple UV patches, a new output directory should be created for each of these patches. The "Output Path" can be a location of choice, but must follow the DPA pipeline rules. For each patch it is necessary to have both a ptc and tex directory within the asset resolution directory. These directories store textures and point clouds generated during the simulation.

Figure 3.8: Rusterizer GUI

The resolution of the cellular automata for the patch is determined by the "Output Resolution" setting in the Rusterizer GUI, rather than the asset directory name. The default setting is 256, creating an output image in a square resolution of 256x256 pixels. If this does not create enough detail for the model, using a resolution of 512 or higher may be necessary. This change increases the simulation time significantly for the patch in question. The choice of resolution is a matter of trial and error. For example, on the model for "QA-ARM-A," there are 8 UV patches. All of these patches use a resolution of 256 except for three patches located on the head, chest and lower torso of the robot which use 512.

The number of frames to be simulated are controlled by the "Start Frame," set by default to a value of 1, and the "End Frame" set by default to 100. The "Start Frame" should always be set to a value of 1 at first, but the "End Frame" can be set to the number of choice. The first 100 frames are often where the most growth occurs on "QA-ARM-A," however the result may vary drastically for other models and input maps.

The "Rusterize!" button in the GUI executes the rusterizer.py script and generates a sequence of images with the .tex extension, as well as point clouds with the .ptc extension. The Maya interface is locked during the simulation, but the user can view the progress and current frame in the command line that was previously used to open Maya. One of the unresolved technical issues with the Rusterizer is that segmentation faults sometimes occur at random frames. This is discussed further in Chapter 6. If the Rusterizer encounters a segmentation fault, the next number after last completed frame in the terminal is the proper "Start Frame" to finish the simulation. For example, if the last completed frame number is 19, then 20 should be set as the new "Start Frame" before the simulation is initiated again. The "End Frame" should remain the same. A segmentation fault is the only reason that the "Start Frame" should be set to a number other than 1. This process can be repeated if necessary.

When the simulation is complete the Rusterizer Shader should be applied to the model, and the simulation output connected to the appropriate place in the shader. This shader is included in the DPA Maya surfacing template file, and will import automatically in the Hypershade window if the file is created using the command "dpacreateworkitem" within the surface workflow. If there are multiple UV patches, a copy of the shader should be created for each patch. Figure 3.9 shows the attributes of the Rusterizer Shader in the Maya interface.

Figure 3.9: Rusterizer Shader Attribute Editor

There are four main input areas for the Rusterizer Shader. The first input field is the location of the Rusterizer output. This would be the "tex" directory that was created prior to running the cellular automata simulation. The next input field is the base filename for the Rusterizer output, or the "Output Base Filename" field. For example, if the filename were "prodShareRusterizerOutput_1001-0014.tex" then the user would type "prodShareRusterizerOutput_1001-0014" in the field.

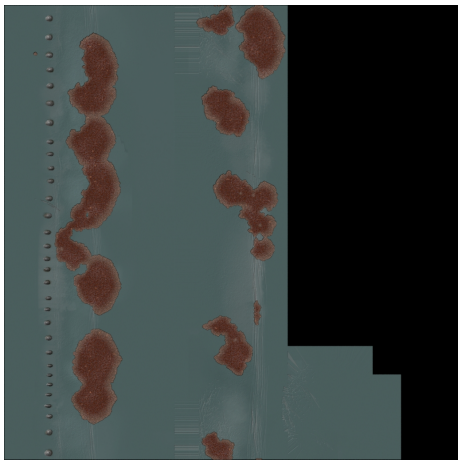
If the current shader is being used for a UV patch outside of the 0-1 range, the Texture Patch U and V Offset fields can be adjusted to correspond to the correct patch location. For example, if a patch exists in the 1-2 U range, and the 0-1 V range, the U offset would be 1 and the V offset would be 0. In Mari, this UV patch would be given the UDIM label "1002."

Base texture paths for a model connect into the "Base Diffuse Map," "Base Bump Map," and "Base Spec Map" fields. The texture for the displacement of the model should be applied to the shading group in Maya. The base diffuse, bump and spec map images need to be converted to the .tex format for the RenderMan shader to be able to use them. To convert an image to the .tex format, the user would navigate to the directory containing the image and type "txmake imageName.extension imageName.tex" in the command line.

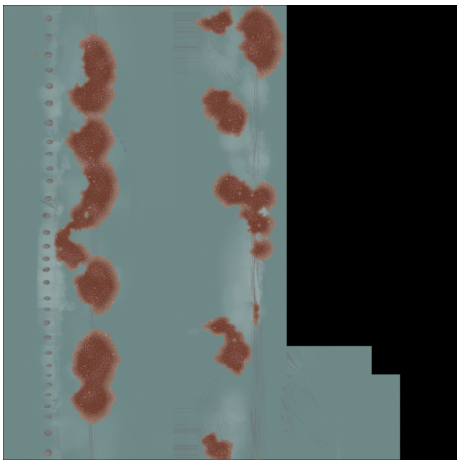
Several presets are included with the Rusterizer Shader. These presets control the rust noise layer colors, as well as the noise scale and frequency values. These settings can all be controlled manually, but the presets will help to give a starting point. The presets can be accessed in Maya by using the "Presets" button near the top of the Attribute Editor.

To animate the growth of the rust, the user must create keyframes in the "Age of Rust" attribute. "Max Life" should match the number of frames that were simulated. For example, if 200 frames have been simulated, "Max Life" would be set to 200 and the user might animate the rust by setting the "Age of Rust" to 1 at the first frame, and setting it to 200 at the last frame.

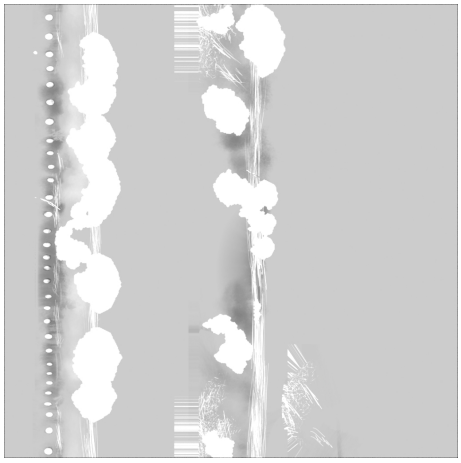
In some cases, the user will want to export the rust result from the shader and paint further details manually in an image editing software like Photoshop or Mari. To do this, the user will select the Map Export Mode from the drop-down selection in the Rusterizer Shader. This attribute is set to "Off" by default. If the mode is set to "Diffuse," "Bump," or "Spec," the shader becomes a surface shader that does not react to light and will display the color of the respective map. The results of the different Map Export Mode settings can be seen in Figure 3.10.



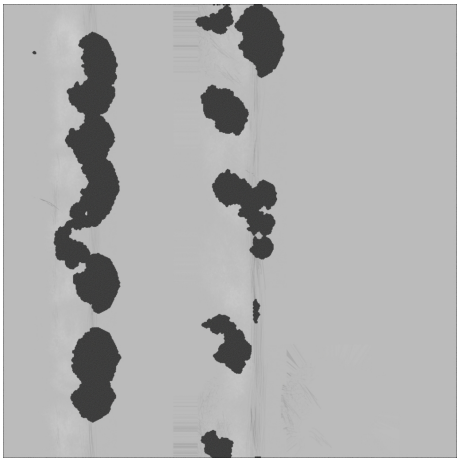
(a) Off



(b) Diffuse



(c) Specular



(d) Bump

Figure 3.10: Rusterizer Shader Map Export Modes

A new render layer should be created in Maya with a square plane facing the camera. The Rusterizer shader should be assigned to this plane, and the Map Export Mode set to the desired map. An image rendered from an orthogonal camera with a square resolution will generate a texture map that can be used as a base for painting. It is also possible to render only the rust by removing the input paths for the base textures in the Rusterizer Shader input fields. The Map Export Mode camera setup is shown in Figure 3.11.

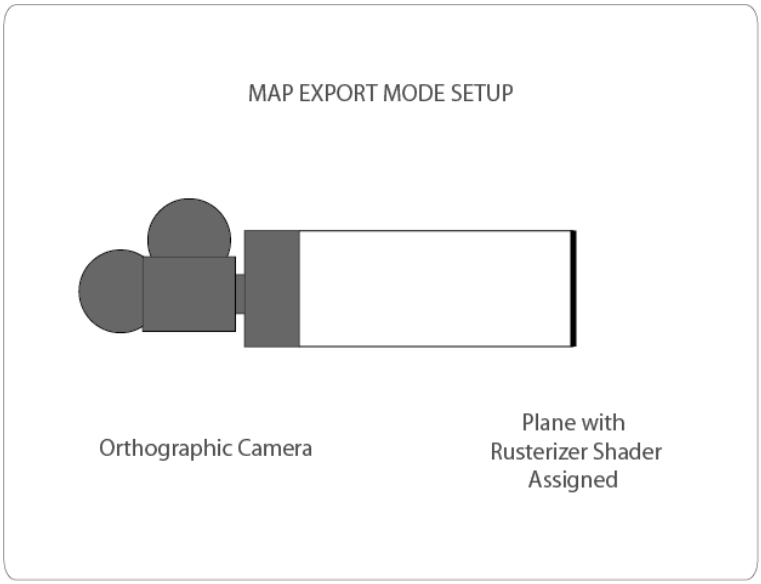


Figure 3.11: Orthographic camera setup for Map Export Mode.

Chapter 4

Implementation

To create the Rusterizer, it became apparent that the RenderMan Shading Language (RSL) would lend itself well to the task due to the procedural aspect of the rust. During research, a method was discovered for using RenderMan in such a way that cellular automata would be feasible. This method required the use of a cellular automata shader in combination with a sequence of RIB, or RenderMan Interface Bytestream files. These files are basically scene descriptions that are used by the renderer to make calculations. Python is used with RenderMan to run the sequence of RIB files and generate point clouds and an image sequence as output. The following is a detailed breakdown of the implementation of the cellular automata simulation and procedural shader used in the workflow.

When the Rusterizer is loaded from the Maya shelf, a script called rusterizerGuiLoader.py creates an instance of a class and calls a function within the class to create an interface. The class exists inside rusterizerGui.py and it connects several functions to the labeled buttons in the GUI. When these buttons are clicked, the respective function is called. For example, when "Rusterize!" is clicked, a function is called that runs rusterizer.py and uses the text field input from the GUI as the appropriate arguments in the script. The GUI layout file, rusterizer.ui, was created using Qt Designer.

The backbone of the Rusterizer tool, rusterizer.py, loops through a specified number of frames to run a series of RIB files. These RIB files call the cellular.sl shader and use it to create point cloud and .tex image file sequences. The script takes the "Rust Distribution Map" and "Rust Probability Map" as input, as well as the location of the cellular.sl shader, output directory, desired resolution, and the start and end frame values.

The RSL cellular automata shader used in the Rusterizer, cellular.sl, was based on Malcolm Kesson's shader by the same name [8]. The rules of the cellular automata were changed, and the resolution of the ouput images was increased. Another data attribute was also included, called "age."

Many different rules were experimented with before a set was chosen for use in the Rusterizer cellular automata simulation. Each of these rules were rendered and compared to determine what parts were working and what parts were not. The first set of rules tested, in Figure 4.1, were very simple and the result showed that the rust edge became very linear as growth progressed. From this test, it was apparent that more variety was necessary. The result is shown in Figure 4.2.

```
if (previous cell state = on)
    current cell state = previous cell state
else if (previous cell state = off AND 3  or 4 neighbors are on)
    current cell state = on
```

Figure 4.1: Rusterizer Cellular Automata Rules, First Test.

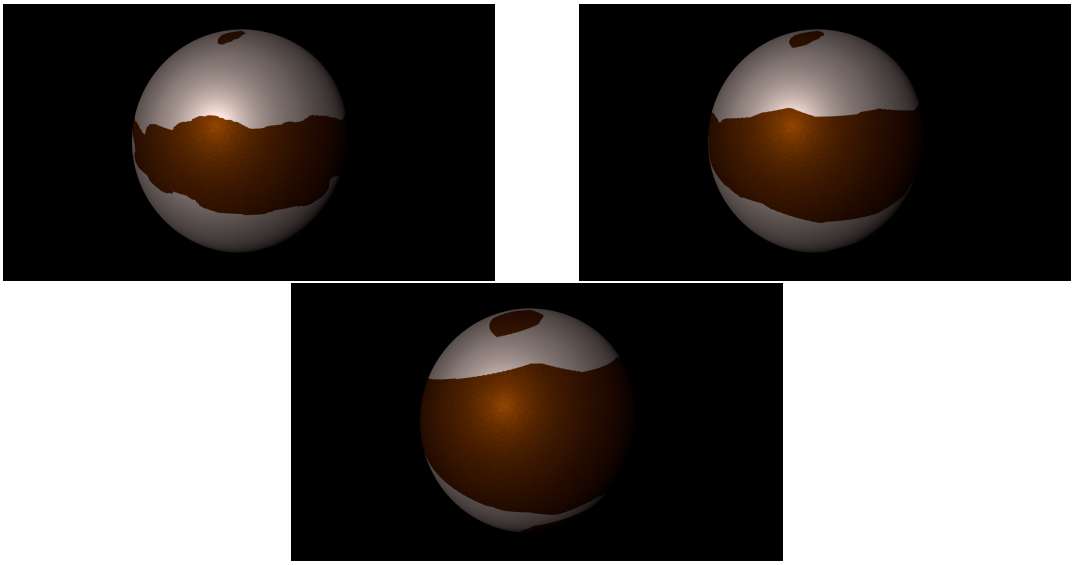


Figure 4.2: Rusterizer Cellular Automata Rules, First Result.

In the next test, variety was added through the addition of probability to the rules, shown in Figure 4.3. Probability was assigned through a random number between zero and one. This resulted in a very nice change in the quality of the edge patterns, as seen in Figure 4.4.

```
if (previous cell state = on)
    current cell state = previous cell state
else if (previous cell state = off AND 3  or 4 neighbors are on AND a random number is >= .5)
    current cell state = on
```

Figure 4.3: Rusterizer Cellular Automata Rules, Second Test.

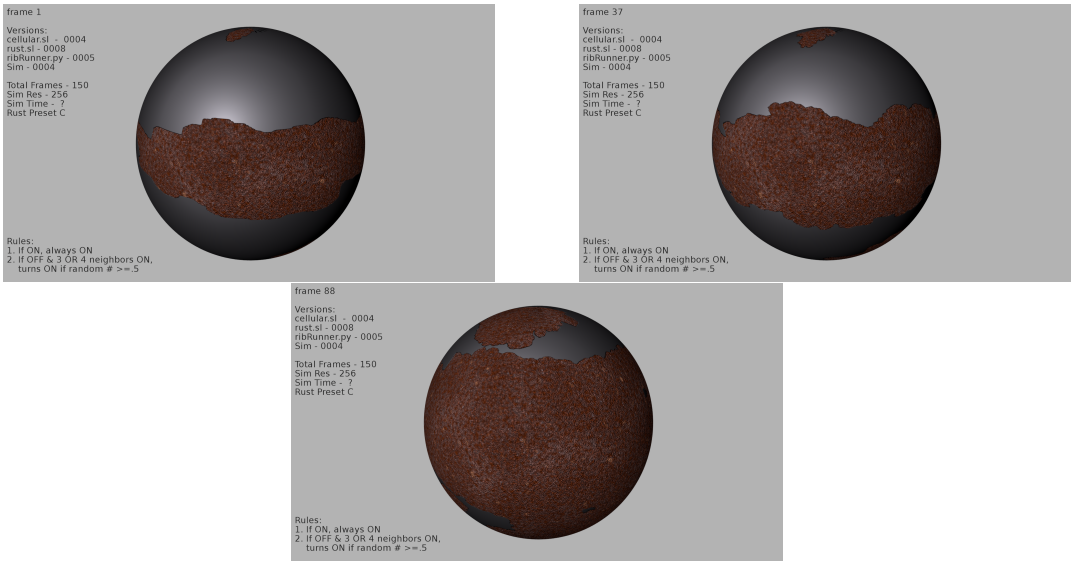


Figure 4.4: Rusterizer Cellular Automata Rules, Second Result.

The previous set of rules was adjusted so that different numbers of neighbors had different probabilities, and this increased the realism of the growth. For the first time, holes started to appear in the patches during growth. An input map was added to set the probability and generate greater variety. This achieved the desired result, but the edge of the rust was still very harsh.

In order to make the transition between rust and the base textures more organic, the procedural shader could assign decreasing opacities of rust to frames several iterations ahead of the current frame. This method became problematic because there was no easy way to smoothly interpolate between different opacity levels, and the harsh transitions looked unnatural, as shown in Figure 4.5. Instead, an "age" value is assigned to points on the surface during the simulation. This works better

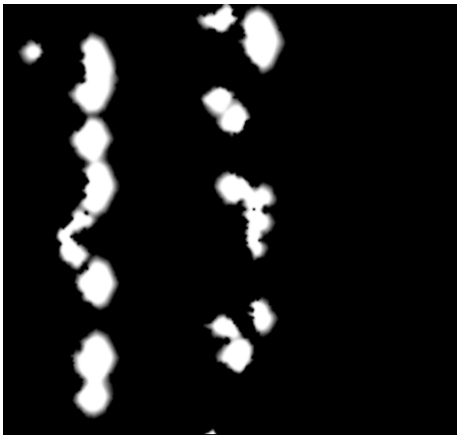
as a method for adding variety to the color of the rust, rather than a fading edge between the base surface texture. In the Rusterizer simulation, the "age" of a point on the surface of a model is stored for each iteration, and is incremented if the previous state of the cell is "on." The result is stored in a new display channel called "_cellAge" in the simulation point cloud output, and is stored in the alpha channel of the .tex image output. As the "age" of a point on the surface increases, the alpha channel value increases towards 1. In the Rusterizer Shader, the color of the rust gets darker as a point ages. There is a maximum level that the "age" can reach, and once it hits this point it never changes. This prevents the rust from continuing to get darker than it should. Figure 4.6 compares the "age" of rust in the alpha channel to the corresponding frame of the simulation.

Figure 4.5: Edge opacity levels.

(a) Frame 1

(b) Frame 1 Alpha Channel

(c) Frame 50



(d) Frame 50 Alpha Channel

(e) Frame 100

(f) Frame 100 Alpha Channel

Figure 4.6: Rusterizer Cellular Automata Simulation.

Using organized point clouds for the simulation reduces the simulation time significantly for the larger patch resolution. For the lower patch resolution the simulation with organized point clouds had a slight increase in time, but this was not very significant. Tables 4.2 and 4.1 show simulation times with and without organized point clouds, for comparison.

Rusterizer Simulation Statistics			
Patch Resolution	**# of Frames**	**Avg. Sim Time Per Frame**	**Total Sim Time**
256x256	100	2.2 seconds	3.7 minutes
512x512	100	9.7 seconds	16.2 minutes

Table 4.1: Rusterizer simulation times without organized point clouds.

Rusterizer Simulation Statistics			
Patch Resolution	**# of Frames**	**Avg. Sim Time Per Frame**	**Total Sim Time**
256x256	100	2.6 seconds	4.3 minutes
512x512	100	5.7 seconds	9.5 minutes

Table 4.2: Rusterizer simulation times with organized point clouds.

The procedural shader rust.sl uses a layered approach. It combines up to six elements: five layers of noise and a solid base color. Each layer of noise can be adjusted with its own color, scale and frequency attributes. Each of the noise layers are composited on top of the layers beneath it, and the number of layers can be adjusted in the Maya attribute editor.

Control for UV patch offsets and a texture map export function allow for textures with multiple UV patches, and to create a base for hand-painted detail. Output images from the cellular automata simulation are used by the shader as a mask that designates whether or not a point will receive rust shading or base texture map shading.

The shader also presents some other important attributes to the user. These attributes include “Age of Rust,” “Number of Noise Layers,” and “Overall Scale.” The Slim file format was used to create a more user-friendly attribute editor [2] that groups different attributes with tabs, as well as adds labels and controls that make sense.

It is important to distinguish between the “age” data attribute in the simulation, and the “Age of Rust” parameter in the Rusterizer Shader. The “Age of Rust” parameter in the Rusterizer Shader determines the frame of the simulation output that is to be used as a mask. The higher the parameter is set, the larger the rust spots will be on the surface.

The Maya file called rusterizerShader.ma loads rust.slo, the compiled version of rust.sl. Four Maya Embedded Language (MEL) scripts save unique attribute presets that can be used to quickly change the color of the rust. These presets were based on colors sampled from images of real rust, and the size and frequency of the noise patterns were also modeled after these examples. It was intended that the presets would be used as starting points for an artist, while allowing manual changes when necessary. These presets can be found in Appendix **??**.

Chapter 5

Results

In order to ensure the usefulness of the Rusterizer for future DPA students, some current DPA students were invited to test the tool. Below is an image generated by another student during the testing phase. Three other students used the tool and gave favorable reviews on it.

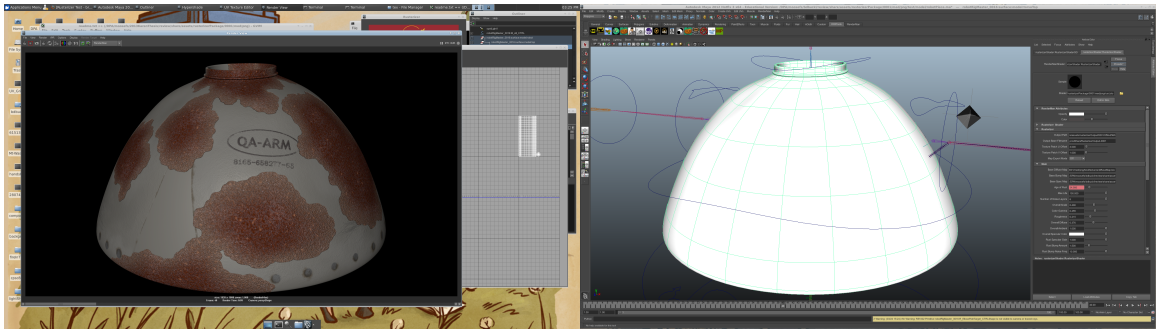


Figure 5.1: Rusterizer student test.

The Rusterizer is highlighted in a short animation titled "Life After QA-ARM-A." This short depicts the sad existence of the robot "QA-ARM-A" after he is laid off from his factory. His eye is broken and his arm is detached, rendering him useless for work in the quality assurance department. He now resides in an alley next to his few belongings and some trash. Still frames from the short are shown in Figure 5.2. The alley scene consists of a single model for the road and the brick wall, a cardboard box, a trash bag, several pieces of crumpled paper, a wood palette, and several models from the short "QA-ARM-A" that were already modeled and surfaced. These pre-existing assets include a hardhat, clipboard, pencil, and the robot.

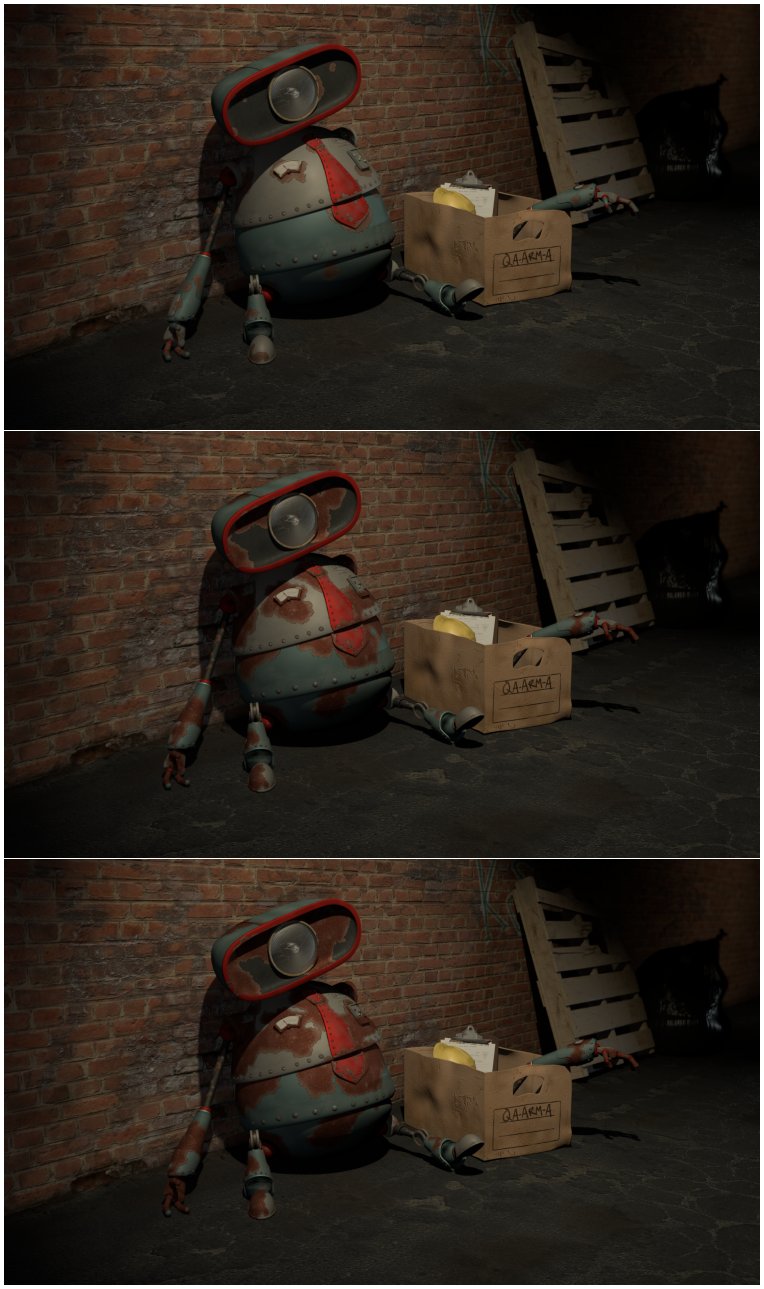


Figure 5.2: Still frames from “Life After QA-ARM-A.”

Chapter 6

Conclusions and Discussion

The Rusterizer is a successful and production-tested surfacing tool for rust. The tool is not perfect, and there are many ways in which future students may want to expand upon the work. The Rusterizer is able to provide interesting animated growth effects for surfaces, and to create static rust patterns on the surface of models. Because the growth can be animated, the artist can choose the frame of the growth that they like best. Control for the color, size and frequency of individual noise patterns was also given to the artist. The Map Export function provides a quick base for manually painted detail, should the art direction in a production call for more specific adjustments.

During testing of the tool, two main technical issues were discovered that have not been resolved. The first problem is the unpredictable segmentation faults that occur during the Rusterizer cellular automata simulation. These segmentation faults do not occur often, and they do not occur at the same frame every time. When the Rusterizer is started up again after a segmentation fault, everything progresses as it should. The issue was replicated in a test of the tool by another student on a different machine.

Second, cells in the simulation occasionally turn "off" after they have already been turned "on." There are no rules in the cellular.sl shader that allow for this to happen, yet the effect can occur. The visual manifestation is that the edge of the rust patches on the surface can sometimes shimmer in a way that is not expected. In most cases, this effect is not very noticeable, although it is in others. It is suspected that there may be some error in the cellular automata shader that is causing this to happen. Further testing and experimentation to deal with this issue would certainly be worthwhile.

There are some relatively simple changes that can be made to the tool to create other surface effects. For example, other shaders could be written to resemble the color of moss, frost, dust or the dying part of a leaf. These shaders can use the simulation output in a similar way to the Rusterizer Shader, but they could also use it in other ways without too much effort.

Another easy change would be to manipulate the cellular automata rules to create different growth patterns, or to grow at a different pace. There are many different types of cellular automata that have been documented and these can all be used as starting points. A good resource for these patterns is conwaylife.com and the accompanying wiki page [6].

In addition to these changes, other more complex improvements would be very nice. The first change that would significantly improve the usefulness of the Rusterizer would be to implement support for Ptex [3]. Ptex, or per-face texture mapping, is a method developed by Walt Disney Animation Studios to allow textures to be applied to models without explicit parameterization. Traditionally, the tedious process of UV layout is necessary before any texture painting can begin. The Ptex approach avoids this step by using the geometry of the model to assign textures. Each polygon receives its own unique identification label and texture. A model using Ptex is shown in Figure 6.1.

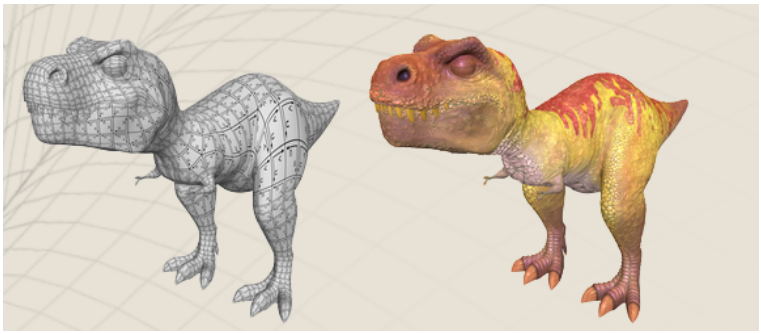


Figure 6.1: Ptex, Walt Disney Animation Studios.

Implementation of Ptex in the Rusterizer would require restructuring the cellular automata simulation so that the cells of the grid exist in three dimensions. The traditional UV coordinates make it relatively simple to loop through a texture and determine which cells are considered neighbors, but this would not be as straightforward with Ptex. Since the geometry of the surface is used to apply textures, the number of neighbors at any point would not be as simple as choosing the pixels on either side of the current point. Some method for interpolating between vertices of the geometry

to assign cell locations would be necessary. The Ptex implementation, once achieved, would allow rust to spread naturally across a surface without having to consider the seams. The feature would also allow a production to create rust surfaces on objects without having to take the time to set up a UV layout.

The cellular automata simulation could benefit from the addition of a random number seed. Currently, the random numbers start from the same point in each simulation. Because of this, the exact same pattern will develop if a simulation were to be run twice. If a production called for a model to be duplicated many times, it would be important for each simulation to have a unique result. RSL does not provide a built-in function for a random seed, so the function would need to be written.

The Rusterizer Shader would be more realistic if it incorporated a "rust drip" layer. This drip pattern was used in the original surfacing for "QA-ARM-A" and can be observed in the real world. The effect occurs when rain water deposits the rust material on the surface below the rust patch. Another simulation would be required to accurately represent this effect over time, and the Rusterizer output could be used to emit particles that travel down the surface of the model. The Rusterizer Shader would need to be modified to include an input location for the "rust drip" simulation output, as well as controls for the color and opacity of the layer. If the "rust drip" effect were only needed for a single frame, this could be hand painted very easily.

More changes that would be useful for the Rusterizer Shader would be the scaling of each attribute so that only the useful or realistic values are available to the artist. Currently, many of the attributes can be scaled infinitely in the positive or negative direction. If these values were scaled so that 0 represented the lowest number that works, and 1 represented the highest number that works, this would help make the Attribute Editor more user friendly. The shader presets help with this issue to some extent.

The Rusterizer GUI provides all the necessary functions, but could be improved with a few more. The first improvement would be the ability for the GUI to run in the background so that the user can continue using Maya during the simulation. It can be tedious to wait for the simulation to finish, and it really is not necessary since rusterizer.py does not require the use of Maya, only RenderMan. Additionally, it would be very helpful if the GUI could automatically display the existing versions of assets, and create new ones if the user does not want to overwrite the old version. A progress bar and a button to cancel or pause the script would also be very nice.

Finally, the ability to specify multiple UV patches that need to be simulated would save a lot of time. If these multiple patches could be distributed to other machines to simulate simultaneously, that would greatly reduce waiting time.

One last recommendation for an extension to the Rusterizer is a script in Maya to automatically set up texture map exporting. The script could create a new render layer, create a new orthogonal camera facing the plane, apply the texture to the surface, select the proper Map Export Mode, and render.

Appendix A

Appendix

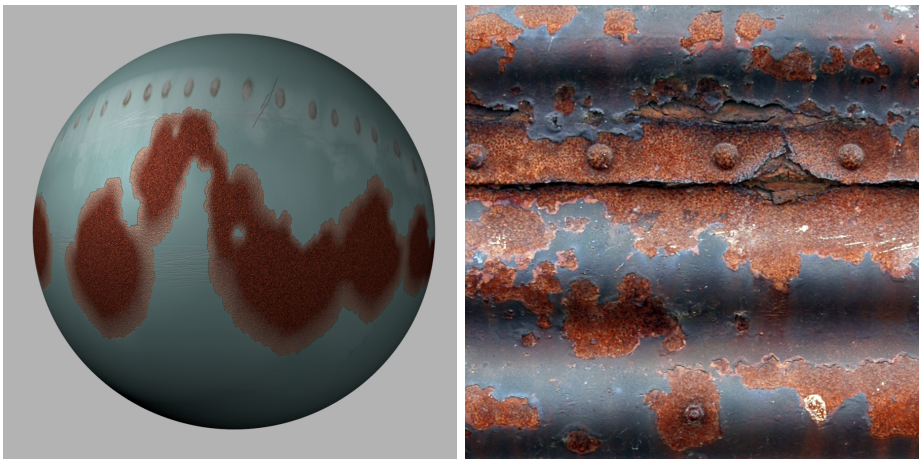


Figure A.1: Preset A and reference image.

Rusterizer Shader Color and Noise Settings: Preset A						
	Color Description	**R**	**G**	**B**	**Scale**	**Frequency**
Base Layer	light orange	.805	.516	.379	-	-
Layer A	light brown	.523	.320	.266	0.8	1.0
Layer B	saturated orange	.632	.285	.168	5.0	1.0
Layer C	deep orange	.535	.207	.164	10.0	1.0
Layer D	dark brown	.293	.184	.141	18.0	1.0
Layer E	darker brown	.207	.094	.102	25.0	1.0

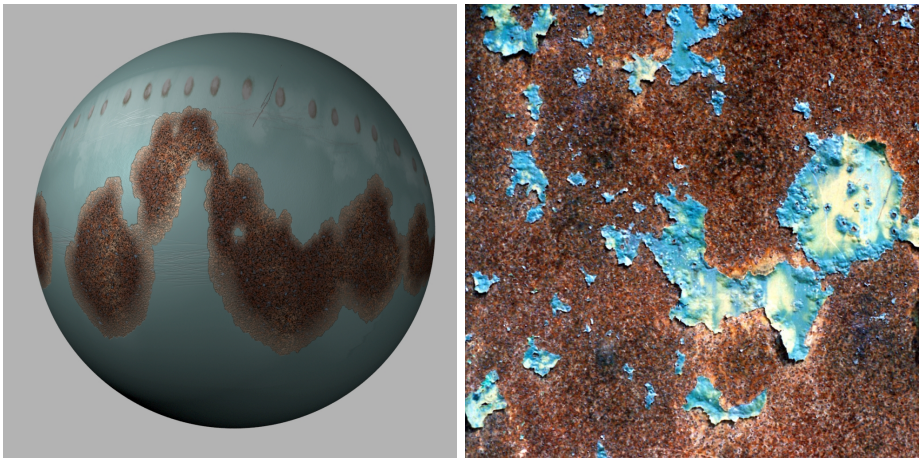


Figure A.2: Preset B and reference image.

Rusterizer Shader Color and Noise Settings: Preset B						
	Color Description	**R**	**G**	**B**	**Scale**	**Frequency**
Base Layer	light yellow	.957	.840	.699	-	-
Layer A	orange	.883	.473	.213	5.0	1.0
Layer B	dark brown	.430	.238	.180	0.5	1.5
Layer C	blue brown	.348	.293	.281	10.0	1.0
Layer D	darkest brown	.074	.094	.070	25.0	1.0
Layer E	light blue	.398	.504	.586	25.0	0.4

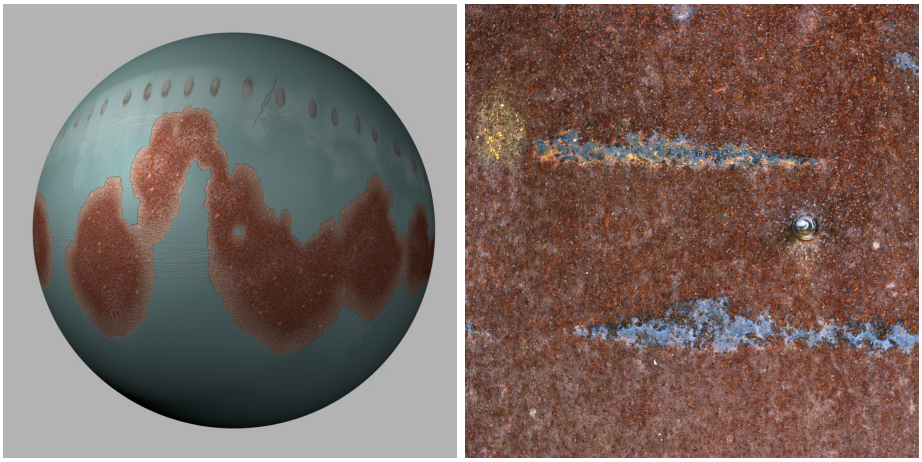


Figure A.3: Preset C and reference image.

Rusterizer Shader Color and Noise Settings: Preset C						
	Color Description	**R**	**G**	**B**	**Scale**	**Frequency**
Base Layer	grey pink	.559	.480	.484	-	-
Layer A	dull coral	.707	.496	.457	5.0	1.0
Layer B	brown	.441	.305	.281	0.5	1.5
Layer C	dark terracotta	.453	.215	.105	10.0	1.0
Layer D	caramel	.809	.535	.402	12.0	0.4
Layer E	light grey pink	.656	.590	.621	35.0	0.4

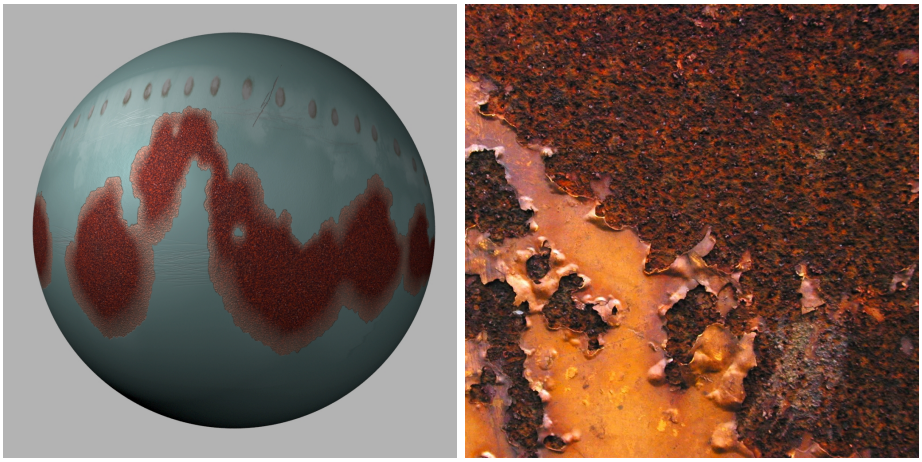


Figure A.4: Preset D and reference image.

Rusterizer Shader Color and Noise Settings: Preset D						
	Color Description	**R**	**G**	**B**	**Scale**	**Frequency**
Base Layer	brown	.492	.313	.262	-	-
Layer A	gold brown	.601	.359	.215	5.0	1.0
Layer B	saturated orange	.789	.254	.074	0.5	1.5
Layer C	brown red	.406	.156	.121	10.0	1.0
Layer D	dark brown	.222	.148	.172	12.0	1.2
Layer E	grey pink	.761	.559	.574	35.0	0.4

Bibliography

[1] Corrosion. `http://search.credoreference.com/content/entry/columency/corrosion/0`, 2013.

[2] Slim File Format. `http://renderman.pixar.com/resources/current/rms/slim_File_Format.html`, 2014.

[3] Brent Burley and Dylan Lacewell. Ptex: Per-Face Texture Mapping for Production Rendering. In *Eurographics Symposium on Rendering 2008*, pages 1155–1164, 2008.

[4] Julie Dorsey and Pat Hanrahan. Modeling and Rendering of Metallic Patinas. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 387–396, New York, NY, USA, 1996. ACM.

[5] Katie Green. The Rust Shader. `http://www.kgreenfx.com/rust/`, 2008.

[6] Nathaniel Johnston. ConwayLife.com: A community for Conway's Game of Life and related cellular automata. `http://www.conwaylife.com/`, 2013.

[7] Paul Kanyuk. RenderMan: An Intro to Pixar's Industry Standard Renderer and Artist Tools. `http://forums.odforce.net/topic/10660-procedural-rust/`, 2009.

[8] Malcolm Kesson. RSL: Cellular Automata. `http://www.fundza.com/rman_shaders/cellular/index.html`, 2002.

[9] Hayden Landis. Production-ready global illumination. *Siggraph course notes*, 16(2002):11, 2002.

[10] David Lipton, Ken Museth, and Ben Sutherland. Jack's Frost: Controllable Magic Frost Simulations for 'Rise of the Guardians'. In *ACM SIGGRAPH 2013 Talks*, SIGGRAPH '13, pages 16:1–16:1, New York, NY, USA, 2013. ACM.

[11] Lauren Perry. RustBucket Report. `http://dropr.com/laurenperry/2078/rustbucket_report/~?p=17990`, 2011.

[12] Joel Schiff. *Cellular Automata*. John Wiley and Sons, Inc., 2008.